

On Checking Skeptical and Ideal Admissibility in Abstract Argumentation Frameworks

Samer Nofal^a, Katie Atkinson^b, and Paul E. Dunne^b

^a*Department of Computer Science, German Jordanian University, Jordan*

^b*Department of Computer Science, University of Liverpool, United Kingdom*

Abstract

Abstract argumentation frameworks (AFs) are directed graphs with vertices being *abstract arguments* and edges denoting the *attacks* between them. Within the context of AFs, we implement and evaluate an algorithm for two essential computational problems: checking *skeptical* and *ideal admissibility*. We evaluate the implemented algorithms using a widely-known benchmark. In terms of the number of solved problem instances and the average running time, our implementation outperforms two prominent systems.

Keywords: directed graph, graph algorithm, argumentation graph, argumentation semantics, computational argumentation

1. Introduction

The notion of *admissibility* (defined shortly) plays a central role in the theory and application of *abstract argumentation frameworks* (AFs) introduced in [11]. For excellent reviews on abstract argumentation research see for example the articles of [3, 16, 4, 5, 9, 24].

An AF is basically a directed graph $G = (V, E)$ such that the vertices of V denote *abstract arguments* and the edges of E represent the *attacks* between them. We say x is a *predecessor* of y (or y is a *successor* of x) if and only if $(x, y) \in E$. For a given set $T \subseteq V$, we denote the set of all predecessors (respectively successors) of the vertices of T by T^- (respectively T^+). A set of vertices $Q \subseteq V$ is *admissible* if and only if $Q^- \subseteq Q^+$ and $Q^+ \cap Q = \emptyset$. A vertex $x \in V$ is *admissible* if and only if x is in an admissible set. A vertex $x \in V$ is *skeptically admissible* if and only if x is contained in every subset-maximal admissible set. A set of vertices $I \subseteq V$ is the *ideal extension* of G if and only if I is the subset-maximal admissible set such that for all $y \in I^-$ y is not admissible. Moreover, a vertex $v \in V$ is *ideally admissible* if and only if v is contained in the ideal extension of G .

In this paper we implement and practically evaluate an *ad hoc* algorithm for checking skeptical and ideal admissibility. The two decision problems are computationally hard in the general case [12, 13]. Take the graph G_1 in figure 1. We note that there are two subset-maximal admissible sets: $\{k, f, b, e\}$ and

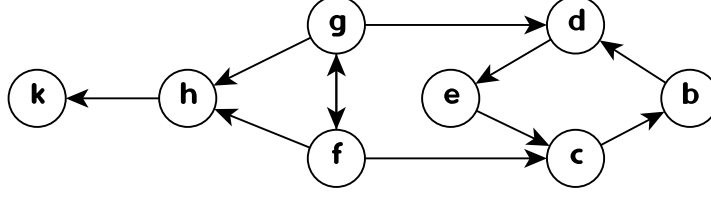


Figure 1: G_1 , an example directed graph.

Algorithm 1: checking skeptical admissibility as in [18].

requires: a directed graph $G = (V, E)$ and a query vertex $x \in V$.
ensures : returns true if x is skeptically admissible; else returns false.

- 1 if x is not admissible then return false;
- 2 if there is an admissible $y \in \{x\}^-$ then return false;
- 3 if there is a subset-maximal admissible set $S \not\supseteq \{x\}$ then return false;
- 4 return true;

$\{k, g, b, e\}$. Thus, the vertices k, b , and e are skeptically admissible; f and g are admissible vertices; b and e are ideally admissible due to the ideal extension $\{b, e\}$.

In section 2 we implement a well-known *ad hoc* algorithm for deciding skeptical admissibility. Then, in section 3 we implement a prevalent *ad hoc* algorithm for checking ideal admissibility. To the best of our knowledge, both algorithms have never yet been fully implemented and empirically evaluated. In section 4 we measure the efficiency of our implementation and observe encouraging indications for its practical use in deciding skeptical and ideal admissibility. We discuss related work in section 5 and lastly conclude the paper in section 6.

2. Checking Skeptical Admissibility

Before introducing the new implementation for checking skeptical admissibility, we recall algorithm 1, which is the *ad hoc* algorithm of the existing implementation considered in [18].

Let us explain the steps of algorithm 1 by checking the admissibility of k in G_1 (figure 1). Initially, we check if there is an admissible set containing k , see line 1 of the algorithm. Subsequently, we find the set of $\{k, f\}$ admissible. Now, we apply line 2 to check if there is an admissible vertex in $\{k\}^-$. Thus, the only vertex in $\{k\}^-$, which is h , is not admissible. Executing line 3, we test every subset-maximal admissible set, which are $\{k, g, b, e\}$ and $\{k, f, b, e\}$. As k is in both sets, we conclude that k is skeptically admissible (line 4).

Referring to algorithm 1, we note that the lines 1 & 2 are computationally easier than line 3 that possibly requires checking a very large number of sets [12].

Algorithm 2: checking skeptical admissibility (Doutre & Mengin [10]).

requires: a directed graph $G = (V, E)$ and a query vertex $x \in V$.
ensures : returns true if x is skeptically admissible; else returns false.

- 1 if x is not admissible then return false;
- 2 if there is an admissible $y \in \{x\}^-$ then return false;
- 3 if G contains no odd cycle then return true;
- 4 if there is a subset-maximal admissible set $S \not\supseteq \{x\}$ then return false;
- 5 return true;

Hence, one might wonder if there is a way to avoid, at least occasionally, the computational burden of line 3. The answer can be found in the *ad hoc* algorithm of Doutre and Mengin [10]. They suggested to check the input graph for *odd cycles* before executing the non-trivial computations of line 3 of algorithm 1. Let $G = (V, E)$ be a directed graph, $G' = (V' \subseteq V, E' \subseteq E)$ be a sub-graph of G with $V' = \{v_0, v_1, v_2, v_3, \dots, v_n\}$, then G' is an *odd cycle* if and only if $|V'|$ is odd, $|V'| = |E'|$, and $E' = \{(v_0, v_1), (v_1, v_2), (v_2, v_3), \dots, (v_{n-2}, v_{n-1}), (v_{n-1}, v_n), (v_n, v_0)\}$. It is known that for directed graphs with no odd cycles the lines 1 & 2 of algorithm 1 are sufficient for deciding skeptical admissibility [6, 14]. However, in the general case, line 3 of algorithm 1 is inevitable. Take the graph G_2 depicted in figure 2. We note that G_2 has two admissible sets: $\{a\}$ and $\{b, d\}$. Although the vertex d is admissible and has no admissible predecessor, d is not skeptically admissible because the admissible set $\{a\} \not\supseteq \{b\}$.

We present algorithm 2, which is the algorithm of Doutre and Mengin introduced in [10]. Algorithm 2 differs from algorithm 1 in checking odd cycles in the input graph, see line 3 of algorithm 2.

For implementing lines 1 and 2 of algorithm 2, we employ the solver of [21]. For performing line 4 of algorithm 2, we run the solver of [20]. Both solvers (i.e. [21] & [20]) are based on the *ad hoc* backtracking algorithms of [19]. For detecting odd cycles (line 3 of algorithm 2), we create algorithm 3. Algorithm 3 is based on the notion of vertex coloring where every vertex in a given graph is assigned a color such that no two adjacent vertices have the same color. By allowing only two colors (say *green* and *blue*) for the vertex coloring, an odd cycle is detected whenever we fail in assigning adjacent vertices different colors. To realize this notion of vertex coloring for odd-cycle detection, we use *white* as the initial color for all vertices in the input graph. Additionally, once we have finished exploring all successors of a given vertex we color it *black*.

To see algorithm 3 in action, let us apply it to G_2 . Initially, apply lines 3 & 4 of algorithm 3 to color every vertex *white*. Then, apply line 6 to color the vertex a *green*, assuming in the first round of the loop at line 5 a is scanned. Then, execute $check(a)$ at line 7. Since $color(b) = white$, at line 14 change the color of b to *blue*. Now, invoke $check(b)$, see line 15. At this point, assume we scan a in the first round of the loop at line 10. Thus, no more actions are taken since $color(a) = green$. In the second round of the loop at line 10, note $color(c) = white$. Subsequently, change the color of c to *green* (line 14) and

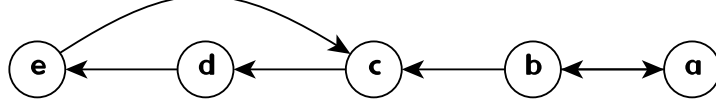


Figure 2: G_2 , an example directed graph.

Algorithm 3: detecting odd cycles.

requires: a directed graph $G = (V, E)$.
ensures : decides whether G contains an odd cycle or not.

```

1 color :  $V \rightarrow \{white, green, blue, black\}$ ;
2 Function detecting-odd-cycles()
3   color  $\leftarrow \emptyset$ ;
4   foreach  $x \in V$  do color  $\leftarrow color \cup \{(x, white)\}$ ;
5   foreach  $x$  with color( $x$ ) = white do
6     color( $x$ )  $\leftarrow green$ ;
7     check( $x$ );
8   no odd cycle in  $G$ ; terminate the program;
9 Function check( $x$ )
10  foreach  $y \in \{x\}^+$  with color( $y$ )  $\neq black$  do
11    if color( $y$ ) = color( $x$ ) then
12      an odd cycle is found; terminate the program;
13    if color( $y$ ) = white then
14      for  $c \in \{green, blue\} \setminus \{color(x)\}$  do color( $y$ )  $\leftarrow c$ ;
15      check( $y$ );
16  color( $x$ )  $\leftarrow black$ ;
```

afterwards invoke *check*(c) (line 15). Therefore, in the only round of the loop at line 10, change the color of d from *white* to *blue* (line 14) and then invoke *check*(d) (line 15). Since *color*(e) = *white*, change the color of e to *green* (line 14) and immediately execute *check*(e) (line 15). As *color*(e) = *color*(c) (line 11), an odd cycle is detected.

3. Checking Ideal Admissibility

We recall algorithm 4, which is the core of the existing *ad hoc* implementation of [17, 19] for checking ideal admissibility. The underlying actions of the algorithm are straightforwardly extracted from the definition of the ideal extension. The issue with algorithm 4 is the huge space possibly required to hold all admissible sets.

Avoiding the intractable space of algorithm 4, Dunne [13] introduced an *ad hoc* algorithm for computing the ideal extension, and subsequently answering the ideal admissibility query on a given vertex. Initially, the algorithm of Dunne

Algorithm 4: checking ideal admissibility as in [17, 19].

requires: a directed graph $G = (V, E)$ and a query vertex $x \in V$.
ensures : returns true if x is ideally admissible; returns false otherwise.

- 1 Let $S = \{S_1, S_2, S_3, \dots, S_n\}$ be the set of all admissible sets of G such that $|S_i| \geq |S_j|$ whenever $i < j$;
- 2 **foreach** $i \in \{1, 2, 3, \dots, n\}$ **do**
- 3 **if** for all $T \in S$ $T^+ \cap S_i = \emptyset$ **then**
- 4 S_i is the ideal extension of G ;
- 5 **if** $x \in S_i$ **then** return true; **else** return false;

Algorithm 5: constructing the ideal extension (Dunne [13]).

requires: a directed graph $G = (V, E)$.
ensures : computes I , the ideal extension of G .

- 1 $S \leftarrow \{x \mid x \text{ is admissible}\}$;
- 2 $S \leftarrow S \setminus S^+$;
- 3 the ideal extension is the subset-maximal $I \subseteq S$ satisfying $I^- \subseteq I^+$;

starts with S being the set of all admissible vertices in the input graph. Then, the algorithm expels from S those vertices that have an admissible predecessor. Lastly, the algorithm finds the ideal extension by computing the subset-maximal admissible set contained in S . Dunne’s algorithm is listed in algorithm 5. Now, apply algorithm 5 to G_1 (figure 1). Perform line 1 to get $S = \{k, f, g, b, e\}$. Then, execute line 2 to obtain $S = \{k, b, e\}$. Thus, the ideal extension is $\{b, e\}$ as $\{b, e\}^- \subseteq \{b, e\}^+$.

A possible bottleneck of Dunne’s algorithm is computing all admissible vertices (line 1). Hence, one might wonder if it is possible to check ideal admissibility without requiring to find *all* admissible vertices. We note that if the query vertex is not admissible or it has an admissible predecessor, then the query vertex is not ideally admissible and so there is no need to pursue computing the admissibility of other vertices. Recall, the basic objective of Dunne’s algorithm is to compute the ideal extension. For this objective, it is likely necessary to compute admissibility for every vertex in the input graph. But here we are concerned answering an ideal admissibility query on a specific vertex. Therefore, taking into consideration the above note on Dunne’s algorithm, we optimize algorithm 5 to algorithm 6 that tests ideal admissibility for a vertex in a given graph. Thus, algorithm 6 checks first the admissibility of the query vertex, say x , and the admissibility of all $y \in \{x\}^-$, see lines 1 & 2. At line 3 the algorithm finds a subset-maximal admissible set S . Then, at line 5 the algorithm removes from S those vertices that have an admissible predecessor. Lastly, the algorithm finds the ideal extension by computing the subset-maximal admissible set contained in S , see line 6. For example, assume we desire to check the ideal admissibility of f in G_1 . Then, we apply algorithm 6. Initially, we check the admissibility of f to find that f is admissible. Afterwards, we check the

Algorithm 6: checking ideal admissibility (optimizing algorithm 5).

requires: a directed graph $G = (V, E)$ and a query vertex $x \in V$.
ensures : returns true if x is ideally admissible; else returns false.

- 1 if x is not admissible then return false;
- 2 if there is an admissible $y \in \{x\}^-$ then return false;
- 3 find a subset-maximal admissible set $S \subseteq V$;
- 4 **if** $x \notin S$ **then** return false;
- 5 $S \leftarrow \{y \in S \mid \text{for all } z \in \{y\}^- \text{ } z \text{ is not admissible}\}$;
- 6 the ideal extension is the subset-maximal $I \subseteq S$ satisfying $I^- \subseteq I^+$;
- 7 **if** $x \in I$ **then** return true **else** return false;

admissibility of g , which is the only predecessor of f . As g is admissible, we conclude that f is not ideally admissible.

As to the implementation of lines 1 & 2 & 5 of algorithm 6, again we run the backtracking-based solver of [21]. Concerning the implementation of line 3, we execute the backtracking-based solver of [20]. With respect to the implementation of line 6, we create algorithm 7.

We describe the notion of algorithm 7. Let $G = (V, E)$ be a directed graph and $S \subseteq V$ be the set containing all admissible vertices that have no admissible predecessor, then algorithm 7 finds the ideal extension of G by removing from S every $x \in S$ with some $y \in \{x\}^-$ such that $\{y\}^- \cap S = \emptyset$. To perform this removal efficiently we employ a counter for each vertex in the graph. The purpose of the counter is to keep track for each vertex the number of its predecessors that are currently in S . If the counter value of some $y \in \{x\}^-$ for a given $x \in S$ is equal to zero, then we remove x from S . Every time a vertex x is removed from S we decrease by one the counter value of every $z \in \{x\}^+$, and then we repeat the same procedure again and again until for all $w \in S$, every predecessor of w has some predecessor in S , which means this condition of admissibility $S^- \subseteq S^+$ is satisfied. Observe that the property of subset maximality is guaranteed because we start with a subset-maximal set S , see line 3 of algorithm 6, and later S might be reduced as described at line 5 of algorithm 6. Then, algorithm 7 computes the ideal extension by removing gradually from S exclusively those vertices that violate the admissibility condition (i.e. $S^- \subseteq S^+$).

Let us now apply algorithm 6 to G_1 to check the ideal admissibility of b . As b is admissible and its only predecessor c is not admissible, we go to line 3 to compute a subset-maximal admissible set. Assume we find $S = \{k, f, b, e\}$. Then, by performing line 5 we obtain $S = \{k, b, e\}$. Therefore, at line 6 we apply algorithm 7 to S . Thus, by the lines 3 & 4 & 5 of algorithm 7 we find $\pi = \{(b, 0), (c, 1), (d, 1), (e, 0), (f, 0), (g, 0), (h, 0), (k, 0)\}$. Applying line 7, we move k from S to \tilde{S} . Hence, $S = \{b, e\}$ and $\tilde{S} = \{k\}$. Then, in the first round of the loop at line 8 we remove k from \tilde{S} , see line 9. As by now $\tilde{S} = \emptyset$, we conclude (line 13) that $S = \{b, e\}$ is the ideal extension. Note, $\{b, e\}^- \subseteq \{b, e\}^+$.

Algorithm 7: implementing line 6 of algorithm 6

requires: a directed graph $G = (V, E)$ with $S \subseteq V$ being the set computed at line 5 of algorithm 6.
ensures : computes the ideal extension of G .

```
1  $\tilde{S} \leftarrow \emptyset$ ;  
2  $\pi : V \rightarrow \{0, 1, 2, \dots, |V|\}$ ;  
3 foreach  $x \in V$  do  $\pi(x) \leftarrow 0$ ;  
4 foreach  $x \in S$  do  
5   foreach  $y \in \{x\}^+$  do  $\pi(y) \leftarrow \pi(y) + 1$ ;  
6 foreach  $x \in S$  do  
7   if there is  $y \in \{x\}^-$  with  $\pi(y) = 0$  then  $S \leftarrow S \setminus \{x\}$ ;  $\tilde{S} \leftarrow \tilde{S} \cup \{x\}$ ;  
8 while  $\tilde{S} \neq \emptyset$  do  
9   remove a vertex  $x$  from  $\tilde{S}$ ;  
10  foreach  $y \in \{x\}^+$  do  
11     $\pi(y) \leftarrow \pi(y) - 1$ ;  
12    if  $\pi(y) = 0$  then  $\tilde{S} \leftarrow \tilde{S} \cup (\{y\}^+ \cap S)$ ;  $S \leftarrow S \setminus \{y\}^+$ ;  
13  $S$  is the ideal extension of  $G$ ;
```

4. Evaluation

The objective of the evaluation is to measure the *average* running-time efficiency of the new implementation of algorithms 2 & 6. The C++ code of our implementation can be found at <https://sourceforge.net/projects/argtools>.

In our evaluation we adopt the settings taken by the second international competition of computational models of argumentation 2017 (ICCMA17) [1]. Thus, we executed our implementation on a machine with an intel-core-i7 processor and four gigabytes of system memory. In contrast, the environment of ICCMA17 is a machine with intel-xeon processor and sixteen gigabyte of system memory such that four gigabytes were allocated for each problem instance. Following ICCMA17, we set a timeout of ten minutes for each problem instance.

As to the benchmark, we considered benchmarks A & D of ICCMA17 for the skeptical and ideal admissibility respectively. Benchmark D is different from A only in the query vertices, but they both consist of the same 350 directed graphs that are distributed into five groups according to computational hardness:

- group 1 includes 50 very easy problem instances.
- group 2 includes 50 easy problem instances.
- group 3 includes 100 medium problem instances.
- group 4 includes 100 hard problem instances.
- group 5 includes 100 very hard problem instances: using 50 directed graphs with two vertices being queried in each one.

Following ICCMA17, we excluded group 1 from evaluation because it includes extremely easy directed graphs, and so we consider 350 problem instances, but using only 300 directed graphs. To evaluate the efficiency of our implementation we compare with two well-known solvers: ArgSemSAT [7] and pyglaf [2]. These solvers are reduction-based where ready-made, back-end systems are executed for solving a reduced form of the problem instance at hand, see the survey of [9] for an overview of reduction-based methods and systems. We selected ArgSemSAT (respectively pyglaf) because it was the best solver in ICCMA17 under the task “check skeptical admissibility” (respectively “check ideal admissibility”).

For the ideal admissibility problem, we report our results in tables 1-5, whereas the tables 6-10 summarize our findings for the skeptical admissibility problem. Note that we report the running times in seconds for only the indicated number of solved instances, which means timeouts are not included within the reported total running times. In the context of experimental algorithms, it is widely accepted to evaluate performance based on the *average* running time or equivalently based on the number of problem instances solved by a given algorithm (solver) within a predefined timeout for each problem instance. These two measures are indeed considered in ICCMA17. Thus, according to ICCMA17, systems are compared against each other by counting the number of problem instances that are solved by the respective systems. If two or more systems solve the same number of problem instances, then ICCMA17 consider the total running time to decide on solver performance. Referring to table 1, we see that the pyglaf solver is more efficient than the new implementation in terms of the number of solved problem instances of group 2 of benchmark D. Nonetheless, depending on the same measure, i.e. the number of solved problem instances, we note that the new implementation *overall* is more efficient than pyglaf, see table 5. Further, the total running time of the new implementation is much less than pyglaf’s total, again see table 5. We stress, the performance of the new implementation is *on average* better than pyglaf although pyglaf outperformed the new implementation in solving some problem instances. Equally important is noting that the number of solved problem instances is a strong indicator for running time efficiency, which is consistent with the measures considered in ICCMA17. Take table 10 for example, if we add the running time of the timeouts (that were solved by the new implementation) to the total running time of ArgSemSAT, then $5092 + ((319 - 309) \times 600) = 11092$ seconds. Obviously, 11092 is much larger than 6405, which is the total running time taken by the new implementation in solving 319 problem instances, again see table 10.

Table 1: checking ideal admissibility for all problem instances in group 2 of benchmark D.

solver	total running time	number of solved problem instances
algorithm 6	563	46
pyglaf	597	50

Table 2: checking ideal admissibility for all problem instances in group 3 of benchmark D.

solver	total running time	number of solved problem instances
algorithm 6	1687	90
pyglaf	2748	94

Table 3: checking ideal admissibility for all problem instances in group 4 of benchmark D.

solver	total running time	number of solved problem instances
algorithm 6	1581	91
pyglaf	6196	76

Table 4: checking ideal admissibility for all problem instances in group 5 of benchmark D.

solver	total running time	number of solved problem instances
algorithm 6	2029	83
pyglaf	1479	64

Table 5: checking ideal admissibility for all problem instances in benchmark D.

solver	total running time	number of solved problem instances
algorithm 6	5860	310
pyglaf	11020	284

Table 6: checking skeptical admissibility for all problem instances in group 2 of benchmark A.

solver	total running time	number of solved problem instances
algorithm 2	1190	50
ArgSemSAT	682	50

Table 7: checking skeptical admissibility for all problem instances in group 3 of benchmark A.

solver	total running time	number of solved problem instances
algorithm 2	2264	98
ArgSemSAT	1239	96

Table 8: checking skeptical admissibility for all problem instances in group 4 of benchmark A.

solver	total running time	number of solved problem instances
algorithm 2	903	87
ArgSemSAT	1367	87

Table 9: checking skeptical admissibility for all problem instances in group 5 of benchmark A.

solver	total running time	number of solved problem instances
algorithm 2	2048	84
ArgSemSAT	1805	76

Table 10: checking skeptical admissibility for all problem instances in benchmark A.

solver	total running time	number of solved problem instances
algorithm 2	6405	319
ArgSemSAT	5093	309

5. Related Work

For the skeptical admissibility problem, we note the previous works that addressed the issue of building *ad hoc* algorithms:

1. the work of Doutre and Mengin [10], which is algorithm 2 that we implemented and empirically evaluated in this paper.
2. the work of Nofal et al [17] that actually follows the high-level structure of the algorithm of Doutre and Mengin [10]. Nevertheless, Nofal et al [17] failed to include in their implementation the proposal of Doutre-Mengin for odd-cycle detection. As discussed earlier odd-cycle detection enhances the efficiency of checking skeptical admissibility.
3. the work of Thang et al [23] is focused on unifying the way of constructing “proofs” for skeptical admissibility among other problems related to the directed graphs of abstract argumentation. We refer the reader to [18] for in-depth comparisons between the work of Thang et al and the work of Doutre and Mengin.
4. the ArgTools system implements the algorithms of [19] that decide skeptical (and ideal) admissibility by basically listing admissible sets. The work of Nofal et al [19] mainly aimed at evaluating a proposed “look-ahead” pruning mechanism.

5. lastly, it is worth noting that heureka [15] and EqArgSolver [22] are *ad hoc* systems implementing the skeptical admissibility problem in addition to other computational problems related to abstract argumentation. In fact, the two systems did not implement the task of checking ideal admissibility. Nonetheless, we are not aware of any published work that elaborates on the specific implementations of heureka and EqArgSolver for solving the skeptical admissibility problem.

As to *ad hoc* algorithms for the ideal admissibility problem, the algorithm of Dunne [13] is optimized and implemented earlier in algorithm 6. We believe that Dunne’s algorithm has never been implemented and empirically evaluated until we did in this work. In fact, the implementation of Nofal et al [17, 19] decides an ideal admissibility by constructing the ideal extension of the input graph entirely through generating a huge list of admissible sets, see algorithm 4.

6. Conclusion

We implemented an *ad hoc* algorithm for deciding skeptical and ideal admissibility within the context of abstract argumentation frameworks. We evaluated our implementation and observed that its performance is superior to two widely-known solvers. A natural direction of this work is to consider parallelizing the algorithms presented in this paper. On the role of parallelism in abstract argumentation, see for example the study of [8]. In general, our findings would motivate further research that focus on developing *ad hoc* algorithms for fundamental computational problems in abstract argumentation. *Ad hoc* algorithms are quite flexible to likely allow for performing efficient computations that fine-tune the use of space and time.

Acknowledgment

We thank the anonymous referees for the comments that improved the presentation of this article.

References

- [1] International competition of computational models of argumentation 2017. <http://argumentationcompetition.org/2017>. Organized by: Sarah A. Gaggl, Thomas Linsbichler, Marco Maratea, and Stefan Woltran.
- [2] Mario Alviano. Ingredients of the argumentation reasoner pyglaf: Python, circumscription, and glucose to taste. In *Proceedings of the 24th RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion 2017*, pages 1–16, 2017.

- [3] Katie Atkinson, Pietro Baroni, Massimiliano Giacomin, Anthony Hunter, Henry Prakken, Chris Reed, Guillermo Ricardo Simari, Matthias Thimm, and Serena Villata. Towards artificial argumentation. *AI Magazine*, 38(3):25–36, 2017.
- [4] Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. An introduction to argumentation semantics. *Knowledge Eng. Review*, 26(4):365–410, 2011.
- [5] T.J.M. Bench-Capon and Paul E. Dunne. Argumentation in artificial intelligence. *Artificial Intelligence*, 171(10):619 – 641, 2007. Argumentation in Artificial Intelligence.
- [6] Claudette Cayrol, Sylvie Doutre, and Jérôme Mengin. On decision problems related to the preferred semantics for argumentation frameworks. *J. Log. Comput.*, 13(3):377–403, 2003.
- [7] Federico Cerutti, Massimiliano Giacomin, and Mauro Vallati. Argsemsat: Solving argumentation problems using SAT. In *Computational Models of Argument - Proceedings of COMMA 2014, Atholl Palace Hotel, Scottish Highlands, UK, September 9-12, 2014*, pages 455–456, 2014.
- [8] Federico Cerutti, Ilias Tachmazidis, Mauro Vallati, Sotirios Batsakis, Massimiliano Giacomin, and Grigoris Antoniou. Exploiting parallelism for hard problems in abstract argumentation. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI’15, pages 1475–1481. AAAI Press, 2015.
- [9] Günther Charwat, Wolfgang Dvorák, Sarah Alice Gaggl, Johannes Peter Wallner, and Stefan Woltran. Methods for solving reasoning problems in abstract argumentation - A survey. *Artif. Intell.*, 220:28–63, 2015.
- [10] Sylvie Doutre and Jérôme Mengin. Preferred extensions of argumentation frameworks: Query answering and computation. In *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, pages 272–288, 2001.
- [11] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358, 1995.
- [12] Paul E. Dunne. Computational properties of argument systems satisfying graph-theoretic constraints. *Artif. Intell.*, 171(10-15):701–729, 2007.
- [13] Paul E. Dunne. The computational complexity of ideal semantics. *Artificial Intelligence*, 173(18):1559 – 1591, 2009.
- [14] Paul E. Dunne and T.J.M. Bench-Capon. Coherence in finite argument systems. *Artificial Intelligence*, 141(1):187 – 203, 2002.

- [15] Nils Geilen and Matthias Thimm. Heureka: A general heuristic backtracking solver for abstract argumentation. In *second international competition on computational argumentation models*, 2017.
- [16] S. Modgil, F. Toni, F. Bex, I. Bratko, C.I. Chesñevar, W. Dvořák, M.A. Falappa, X. Fan, S.A. Gaggl, A.J. García, M.P. González, T.F. Gordon, J. Leite, M. Možina, C. Reed, G.R. Simari, S. Szeider, P. Torroni, and S. Woltran. The added value of argumentation. In Sascha Ossowski, editor, *Agreement Technologies*, volume 8 of *Law, Governance and Technology Series*, pages 357–403. Springer Netherlands, 2013.
- [17] Samer Nofal, Katie Atkinson, and Paul E. Dunne. Algorithms for argumentation semantics: Labeling attacks as a generalization of labeling arguments. *J. Artif. Intell. Res. (JAIR)*, 49:635–668, 2014.
- [18] Samer Nofal, Katie Atkinson, and Paul E. Dunne. Algorithms for decision problems in argument systems under preferred semantics. *Artif. Intell.*, 207:23–51, 2014.
- [19] Samer Nofal, Katie Atkinson, and Paul E. Dunne. Looking-ahead in backtracking algorithms for abstract argumentation. *Int. J. Approx. Reasoning*, 78:265–282, 2016.
- [20] Samer Nofal, Katie Atkinson, and Paul E. Dunne. A system for generating subset-maximal admissible sets of abstract argumentation frameworks. [https://sourceforge.net/projects/argtools/files/Generate all preferred extensions/](https://sourceforge.net/projects/argtools/files/Generate_all_preferred_extensions/), June 2018.
- [21] Samer Nofal, Katie Atkinson, and Paul E. Dunne. A system for deciding admissibility in abstract argumentation frameworks. [https://sourceforge.net/projects/argtools/files/Decide Credulous Acceptance \(admissible\)/](https://sourceforge.net/projects/argtools/files/Decide_Credulous_Acceptance_(admissible)/), May 2018.
- [22] Odinaldo Rodrigues. A forward propagation algorithm for the computation of the semantics of argumentation frameworks. In *Theory and Applications of Formal Argumentation - 4th International Workshop, TAFA 2017, Melbourne, VIC, Australia, August 19-20, 2017, Revised Selected Papers*, pages 120–136, 2017.
- [23] Phan Minh Thang, Phan Minh Dung, and Nguyen Duy Hung. Towards a common framework for dialectical proof procedures in abstract argumentation. *J. Log. Comput.*, 19(6):1071–1109, 2009.
- [24] Matthias Thimm and Serena Villata. The first international competition on computational models of argumentation: Results and analysis. *Artif. Intell.*, 252:267–294, 2017.